

Programming Design

Algorithms and Recursion

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Introduction

- It is said that:
 - **Programming = Data structure + Algorithms.**
 - [http://en.wikipedia.org/wiki/Algorithms %2B Data Structures %3D Programs](http://en.wikipedia.org/wiki/Algorithms_%2B_Data_Structures_%3D_Programs)
 - To design a program, choose data structures to store your data and choose algorithms to process your data.
- Each of “data structures” and “algorithms” requires one (or more) courses.
 - We will only give you very basic ideas.

Algorithms

- Today we talk about **algorithms**, collections of steps for completing a task.
 - In general, an algorithm is used to **solve a problem**.
 - The most common strategy is to divide a problem into small pieces and then solve those **subproblems**.
 - We will introduce **recursion**, a way to solve a problem based on the solution/outcome of subproblems.
- For a problem, there may be multiple algorithms.
 - The first criterion, of course, is **correctness**.
 - **Time complexity** is typically the next for judging correct algorithms.
- As examples, we introduce two specific problems: **searching** and **sorting**.
- Let's watch a **video**!

Outline

- **Recursion**
- Searching
- Sorting

Recursive functions

- A function is **recursive** if it invokes itself (directly or indirectly).
- The process of using recursive functions is called **recursion**.
- Why recursion?
 - Many problems can be solved by dividing the original problem into one or several smaller pieces of **subproblems**.
 - Typically subproblems are **quite similar** to the original problem.
 - With recursion, we write one function to solve the problem by **using the same function** to solve subproblems.

Example 1: finding the maximum

- Suppose that we want to find the maximum number in an array $A[1..n]$ (which means A is of size n).
 - Is there any subproblem whose solution can be utilized?
 - Subproblem: Finding the maximum in an array with size smaller than n .
- A strategy:
 - Subtask 1: First find the maximum of $A[1..(n-1)]$.
 - Subtask 2: Then compare that with $A[n]$.
- How would you visualize this strategy?
- While subtask 2 is simple, subtask 1 is **similar** to the original task.
 - It can be solved with the **same** strategy!

Example 1: finding the maximum

- Let's try to implement the strategy.
- First, I know I need to write a function whose header is:

```
double max(double array[], int len);
```

- This function returns the maximum in **array** (containing **len** elements).
- I **want** this to happen, though at this moment I do not know how.
- Now let's implement it:
 - If the function **really works**, subtask 1 can be completed by invoking

```
double subMax = max(array, len - 1);
```

- Subtask 2 is done by comparing **subMax** and **array[len - 1]**.

Example 1: finding the maximum

- A (wrong) implementation:
- What will happen if we really invoke this function?
 - The program will not terminate!
 - Even when **len** is 1 in an invocation, we will still try to invoke **max(array, 0)**.
- For an array whose size is 1:
 - That number is the maximum!
- With this, we can add a **stopping condition** into our function.

```
double max(double array[], int len)
{
    double subMax = max(array, len - 1);
    if (array[len - 1] > subMax)
        return array[len - 1];
    else
        return subMax;
}

int main()
{
    double a[5] = {5, 7, 2, 4, 3};
    cout << max(a, 5);
    return 0;
}
```

Example 1: finding the maximum

- A correct implementation is:
- What is the outcome?

```
int main()
{
    double a[5] = {5, 7, 2, 4, 3};
    cout << max(a, 5);
    return 0;
}
```

- Both **else** can be removed. Why?

```
double max (double array[], int len)
{
    if (len == 1) // stopping condition
        return array[0];
    else
    {
        // recursive call
        double subMax = max (array, len - 1);
        if (array[len - 1] > subMax)
            return array[len - 1];
        else
            return subMax;
    }
}
```

Example 1: finding the maximum

- Is it okay to remove both **else**? Why?

```
double max (double array[], int len)
{
    if (len == 1) // stopping condition
        return array[0];
    else
    {
        // recursive call
        double subMax = max (array, len - 1);
        if (array[len - 1] > subMax)
            return array[len - 1];
        else
            return subMax;
    }
}
```

```
double max (double array[], int len)
{
    if (len == 1) // stopping condition
        return array[0];
    // recursive call
    double subMax = max (array, len - 1);
    if (array[len - 1] > subMax)
        return array[len - 1];
    return subMax;
}
```

Example 2: computing factorials

- How to write a function that computes the factorial of n ?
 - A subproblem: computing the factorial of $n - 1$.
 - A strategy: First calculate the factorial of $n - 1$, then multiply it with n .

```
int factorial (int n)
{
    if (n == 1) // stopping condition
        return 1;
    else
        // recursive call
        return factorial (n - 1) * n;
}
```

Example 2: computing factorials

- When we invoke this function with argument 4:
- **factorial(4)**
= **factorial(3) * 4**
= **(factorial(2) * 3) * 4**
= **((factorial(1) * 2) * 3) * 4**
= **((1 * 2) * 3) * 4**
= **(2 * 3) * 4**
= **6 * 4**
= **24**

Example 3: the Fibonacci sequence

- Write a recursive function to find the n th Fibonacci number.
 - The Fibonacci sequence is 1, 1, 2, 3, 5, 8, 13, 21, Each number is the sum of the two preceding numbers.
 - The n th value can be found once we know the $(n - 1)$ th and $(n - 2)$ th values.

```
int fib (int n)
{
    if (n == 1)
        return 1;
    else if (n == 2)
        return 1;
    else // two recursive calls
        return (fib (n - 1) + fib (n - 2));
}
```

Some remarks

- There must be a **stopping condition** in a recursive function. Otherwise, the program will not terminate.
- In many cases, a recursive strategy can also be implemented with **loops**.
 - E.g., writing a loop for finding a maximum and factorial.
 - But sometimes it is hard to use loops to imitate a recursive function.
- Compared with an equivalent iterative function, a recursive implementation is usually **simpler** and **easier to understand**.
- However, it generally uses **more memory spaces** and is **more time-consuming**.
 - Invoking functions has some cost.

Complexity issue of recursion

- In some cases, recursion is efficient enough.
 - E.g., finding a maximum or calculating the factorial.
- In some cases, however, recursion can be very **inefficient!**
 - E.g., Fibonacci.
- Let's compare the efficiency of two different implementations.

Complexity issue of recursion

- Two implementations:

```
int fib (int n)
{
    if (n == 1)
        return 1;
    else if (n == 2)
        return 1;
    else // two recursive calls
        return (fib (n-1) + fib (n-2));
}
```

```
double fibRepetitive (int n)
{
    if (n == 1)
        return 1;
    else if (n == 2)
        return 1;
    double* fib = new double[n];
    fib[0] = 1;
    fib[1] = 1;
    for (int i = 2; i < n; i++)
        fib[i] = fib[i - 1] + fib[i - 2];
    double result = fib[n - 1];
    delete[] fib; // to be explained
    return result;
}
```

Complexity issue of recursion

- Which one is faster?

```
int main ()
{
    int n = 0;
    cin >> n;
    cout << fibRepetitive(n) << "\n"; // algorithm 1
    cout << fib(n) << "\n"; // algorithm 2
    return 0;
}
```

Polynomial time vs. exponential time

- Given n :
 - The repetitive way has around c_1n steps, where $c_1 > 0$ is a constant.
 - The recursive way has around c_22^n steps, where $c_2 > 0$ is a constant.
- When n is large enough, c_22^n is much larger than c_1n .
 - Even if $c_1 \ll c_2!$
 - We say the repetitive way is **more efficient**.
- Technically, we say that:
 - The repetitive way is a **polynomial-time** algorithm
 - The recursive way is an **exponential-time** algorithm.
- In general, an exponential-time algorithm is just too inefficient.

Power of recursion

- Though recursion is sometimes inefficient, typically implementation is easier.
- Let's consider the classic example “**Hanoi Tower**”.
 - There are three pillars and disks of different sizes which can slide onto any pillar. Disc i is smaller than disc j if $i < j$.
 - A large disc cannot be placed on top of a small disc.
- Initially, all discs are at pillar A. We want to move them to pillar C:
 - Only one disk can be moved at a time.
 - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack.
- Let's watch a **video**!
- What are the steps that solve the Hanoi Tower problem in the fastest way?

Power of recursion

- A recursive implementation:

```
void hanoi (char from, char via, char to, int disc)
{
    if (disc == 1)
        cout << "From " << from << " to " << to << endl;
    else
    {
        hanoi (from, to, via, disc - 1);
        cout << "From " << from << " to " << to << endl;
        hanoi (via, from, to, disc - 1);
    }
}
```

```
#include <iostream>
using namespace std;

int main ()
{
    int disc = 0; // number of discs
    cin >> disc;
    char a = 'A', b = 'B', c = 'C';

    hanoi (a, b, c, disc);

    return 0;
}
```

- Is there a good way of solving the Hanoi Tower problem iteratively?

Outline

- Recursion
- **Searching**
- Sorting

Searching

- One fundamental task in computation is to **search** for an element.
 - We want to determine whether an element exists in a set.
 - If yes, we want to locate that element.
 - E.g., looking for a string in an article.
- Here we will discuss how to search for an integer in an one-dimensional array.
- Whether the array is **sorted** makes a big difference.

Searching

- Consider an integer array $A[1..n]$ and an integer p .
- How to determine whether p exists in A ?
- If so, where is it?
 - Assume that we only need to find one p even if there are multiple.
- Suppose that the array is unsorted.
- One of the most straightforward way is to apply a **linear search**.
 - Compare each element with p **one by one**, from the first to the last.
 - Whenever we find a match, report its location.
 - Conclude that p does not exist if we end up with nothing.
- The number of operations we need to execute is roughly proportional to n .

Binary search

- What if the array is sorted?
- We may still apply the linear search.
- However, we may improve the efficiency by implementing a **binary search**.
 - First, we compare p with the median m (e.g., $A[(n + 1) / 2]$ if n is odd).
 - If p equals m , bingo!
 - If $p < m$, we know p must exist in **the first half** of A if it exists.
 - If $p > m$, we know p must exist in **the second half** of A if it exists.
 - For the latter two cases, we will continue searching in the **subarray**.
- Let's watch a **video**!

Binary search: pseudocode

- binarySearch(a sorted array A , search in between $from$ and to , search for p)
 - if* $n = 1$
 - return* true if $A_{from} = p$; *return* false otherwise
 - else*
 - let $median$ be floor($(from + to) / 2$)
 - if* $p = A_{median}$
 - return* true
 - else if* $p < A_{median}$
 - return* binarySearch(A , $from$, $median$, p)
 - else*
 - return* binarySearch(A , $median + 1$, to , p)

Linear search vs. binary search

- In binary search, the number of instructions to be executed is roughly proportional to $\log_2 n$.
- So binary search is **much more efficient** than linear search!
 - The difference is huge if the array is large.
 - However, binary search is possible only if the array is sorted.
 - Is it worthwhile to sort an array before we search it?
- It is natural to implement binary search with **recursion**.
 - A subproblem is to search for the element in one half of the array.
- Binary search can also be implemented with repetition.
 - Is it natural to do so?

Outline

- Recursion
- Searching
- **Sorting**

Sorting

- Given a one-dimensional integer array A of size n , how to sort it?
- Given numbers 6, 9, 3, 4, and 7, how would you sort them?
- Recall what you typically do when you play poker:
 - First put the first number 6 aside.
 - Compare the second number 9 with 6. Because $9 > 6$, put 9 to the right of 6.
 - Compare the third number 3 with the **sorted list** (6, 9). Because $3 < 6$, put 3 to the left of 6.
 - Compare 4 with (3, 6, 9). Because $3 < 4 < 6$, **insert** 4 in between 3 and 6.
 - Compare 7 with (3, 4, 6, 9). Because $6 < 7 < 9$, insert 7 in between 6 and 9.
 - The result is (3, 4, 6, 7, 9).
- Let's watch a **video**!

Insertion sort

- The above algorithm is called **insertion sort**.
 - The key is to maintain a sorted list.
 - Then for each number in the unsorted list, **insert** it into the proper location so that the sorted list **remains sorted**.
- How would you implement the insertion sort?
 - Recursion or repetition?
 - If recursion, what is your strategy?

(Non-repetitive) insertion sort

- insertionSort(a non-repetitive array A , the array length n , an index $cutoff < n$)
 // at any time, $A_{1..cutoff}$ is sorted and $A_{(cutoff+1)..n}$ is unsorted
 if $A_{cutoff+1} < A_{1..cutoff}$
 let p be 1
 else
 find p such that $A_{p-1} < A_{cutoff+1} < A_p$
 insert $A_{cutoff+1}$ to A_p and shift $A_{p..cutoff}$ to $A_{(p+1)..(cutoff+1)}$
 if $cutoff + 1 < n$
 insertionSort($A, n, cutoff + 1$)
- What if A is repetitive?

Insertion sort

- Roughly how many instructions do we need for insertion sort?
 - We need to do n insertions.
 - To insert the k th value, we search for a position and shift some elements.
 - A linear search: at most k comparisons.
 - Shifting: at most k shifts.
 - Roughly we need $1 + 2 + \dots + n$ operations, which is proportional to n^2 .
- Does binary search help?

Mergesort (Merge sort)

- Insertion sort is **simple** and fast!
 - Not really “fast”, but faster than many similar sorting algorithm.
 - Because its idea and implementation is simple, it is faster than most algorithms when the array size is **small**.
- Interestingly, there is another sorting algorithm:
 - Its idea is somewhat similar to insertion sort.
 - But it is significantly faster for large arrays!
- This algorithm is called **mergesort**.

Mergesort (Merge sort)

- Recall that in an insertion sort, we need to insert one number into a sorted list for many times.
- A key observation is that “inserting” **another sorted list** of size k into a sorted list can be faster than inserting k separate numbers one by one!
 - So such “inserting” is actually “**merging**”.
- Given an unsorted array, we will:
 - First split the array into two parts, the first half and second half.
 - Then sort each subarray.
 - Finally, merge these two subarrays.
- Mergesort is perfect for recursion!

Mergesort (Merge sort): pseudocode

mergeSort(an array A , the array length n)

let $median$ be $\text{floor}((1 + n) / 2)$

mergeSort($A_{1..median}$, $median$) // now $A_{1..median}$ is sorted

mergeSort($A_{(median + 1)..n}$, $n - median + 1$) // now $A_{(median + 1)..n}$ is sorted

merge $A_{1..median}$ and $A_{(median + 1)..n}$ // how?

Mergesort (Merge sort)

- Interestingly, insertion sort is a special way of running mergesort.
 - Not splitting the array into two halves.
 - Instead, splitting it into $A[1..n - 1]$ and $A[n]$.
- Once we use the “smart split”, the **efficiency** is improved a lot!
 - Insertion sort: Roughly proportional to n^2 .
 - Merge sort: Roughly proportional to $n \log n$.
- A simple observation can make a huge difference!